


**PATENT**  
**5681-62301**  
**P9307**

"EXPRESS MAIL" MAILING LABEL  
NUMBER EL990142769US  
DATE OF DEPOSIT JANUARY 29, 2004  
I HEREBY CERTIFY THAT THIS PAPER OR  
FEE IS BEING DEPOSITED WITH THE  
UNITED STATES POSTAL SERVICE  
"EXPRESS MAIL POST OFFICE TO  
ADDRESSEE" SERVICE UNDER 37 C.F.R.  
§1.10 ON THE DATE INDICATED ABOVE  
AND IS ADDRESSED TO THE  
COMMISSIONER FOR PATENTS, BOX  
PATENT APPLICATION, P.O. Box 1450,  
ALEXANDRIA, VA 22313-1450

  
Derrick Brown

PARALLEL TEXT EXECUTION ON LOW-END EMULATORS AND DEVICES

By:

Yaniv Vakrat and Victor Rosenman

B. Noel Kivlin  
Meyertons, Hood, Kivlin, Kowert & Goetzel  
P.O. Box 398  
Austin, TX 78767-0398

## Parallel Text Execution on Low-End Emulators and Devices

### CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims the benefit of Provisional Application No. 60/443,795. This application is related to Application No. (STC File No. 47900), entitled, *Automated Test Execution Framework with Central Management*.

### BACKGROUND OF THE INVENTION

#### 1. Field of the Invention.

[0002] The present invention relates generally to hardware and software testing and verification, and specifically to testing software on low-end emulators and computing devices.

#### 2. Description of the Related Art.

[0003] The meanings of acronyms and certain terminology used herein are given in Table 1:

Table 1

API	Application programming interface
CLDC	Connected, limited device configuration. CLDC is suitable for devices with 16/32-bit RISC/CISC microprocessors/controllers, having as little as 160 KB of total memory available.
HTTP	HyperText Transfer Protocol
ID	Identifier
IP	Internet Protocol
J2EE	Java 2 Enterprise Edition
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
JAD	Java application descriptor
JAM tags	Mandatory fields in a JAD file
JAR	Java archive
MIDlet	A MIDP application
MIDP	Mobile information device profile. A set of Java APIs, which, together with the CLDC, provides a complete J2ME application runtime

	environment targeted at mobile information devices.
--	---

[0004] MIDP is defined in Mobile Information Device Profile (JSR-37), JCP Specification, Java 2 Platform, Micro Edition, 1.0a (Sun Microsystems Inc., Palo Alto, California, December 2000). MIDP builds on the Connected Limited Device Configuration (CLDC) of the Java 2 Platform, Micro Edition (J2ME) (available from Sun Microsystems Inc., Palo Alto, California). The terms Sun, Sun Microsystems, Java, J2EE, J2ME, J2SE, and the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States of America and other countries. All other company and product names may be trademarks of their respective companies. A portion of the disclosure of this patent document contains material that is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by any one of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

[0005] Tools have been developed in recent years to aid in the design verification of hardware and software systems, for example software suites, hardware circuitry, and programmable logic designs. In order to assure that the design complies with its specifications, it is common to generate a large number of input or instruction sequences to assure that the design operates as intended under a wide variety of circumstances. In general, test systems produce a report indicating whether tests have been passed or failed, and, in some cases may even indicate a module that is estimated to be faulty.

[0006] Conventionally, to test a device under development (such as a mobile information device), or to test software designed to run on such a device, a developer connects the de-

vice to an appropriate test system. The target device under test may be connected to the test system either directly or via a communication emulator. The developer selects a battery of test programs to run on the target device while monitoring its behavior. Running the complete battery of tests can commonly take many hours or even days. This problem is particularly acute in testing low-end computing devices, such as cellular telephones and other mobile information devices, which have limited computing power and memory resources. Thus, testing on the target device can become a serious bottleneck in the development cycle.

#### **SUMMARY OF THE INVENTION**

[0007] Embodiments of the present invention provide methods and systems for parallel testing of multiple low-end computing devices, such as mobile information devices. Multiple computing devices are connected to a test server, either directly or via an emulator. Each of the devices is assigned a unique identifier (ID), which allows the server to keep track of which tests have been assigned to and carried out by each device. Whenever a device completes a test (or a bundle of tests), it reports the results to the server and requests the next test to execute, using its unique identifier in the messages that it sends to the server. Based on the unique identifier and the report, the server selects the next test or test bundle to assign to this device. This mechanism enables the server to balance and track the load of testing among an arbitrarily large number of client devices, and thus to complete the test suite in far less time than is required by test systems known in the art.

[0008] The invention provides a method for testing computing devices, which is carried out by providing a suite of test programs on a server for execution by a plurality of the

computing devices that are coupled to the server, assigning a respective unique identifier to each of the plurality of the computing devices for use in communicating with the server, downloading the test programs from the server for execution by  
5 the computing devices coupled thereto, so that at least first and second computing devices among the plurality execute different first and second test programs from the suite substantially simultaneously. The method further includes receiving  
10 messages at the server from the computing devices with respect to the execution of the test programs, each of the messages containing the respective unique identifier, and controlling the execution of the first and second test programs in the suite based on the messages.

[0009] According to one aspect of the method, the computing devices are MIDP-compliant devices, and the test programs are MIDlets, which are packaged in respective JAD files and JAR files, and the method includes downloading the JAD files and the JAR files to the MIDP-compliant devices.  
15

[0010] Yet another aspect of the method includes evaluating the JAD files, wherein the JAR files are downloaded responsively to the evaluation of the JAD files.  
20

[0011] According to another aspect of the method, at the test program comprises a bundle of tests, and requests are received from the computing devices to determine a next test to execute in the bundle. Responsively to a selection at the server, based on the respective unique identifier contained in the requests, a determination is made of the next test to execute on each of the computing devices, and messages are sent from the server to the computing devices indicating the selection.  
25  
30

[0012] According to a further aspect of the method, the respective unique identifier of each of the computing devices includes an IP address.

[0013] According to yet another aspect of the method, assigning the respective unique identifier includes receiving an initial request from each of the computing devices to download one of the test programs, and assigning the respective  
5 unique identifier in response to the initial request.

[0014] According to still another aspect of the method, the computing devices are coupled to the server via a common test host, wherein an identifier of the common test host is shared by each of the computing devices in the respective  
10 unique identifier thereof.

[0015] The invention provides a computer software product, including a computer-readable medium in which computer program instructions are stored, which instructions, when read by a computer, cause the computer to perform a method for testing  
15 computing devices, which is carried out by providing a suite of test programs on a server for execution by a plurality of the computing devices that are coupled to the server, assigning a respective unique identifier to each of the plurality of the computing devices for use in communicating with the  
20 server, downloading the test programs from the server for execution by the computing devices coupled thereto, so that at least first and second computing devices among the plurality execute different first and second test programs from the suite substantially simultaneously. The method further includes re-  
25 ceiving messages at the server from the computing devices with respect to the execution of the test programs, each of the messages containing the respective unique identifier, and controlling the execution of the first and second test programs in the suite based on the messages.

[0016] The invention provides a server for testing computing devices, including a communication interface for coupling a plurality of the computing devices thereto, such that a  
30 respective unique identifier is assigned to each of the plural-

ity of the computing devices for use in communicating with the server via the communication interface. The server is adapted to provide a suite of test programs for execution by the computing devices that are coupled to the server, and to download  
5 the test programs via the communication interface for execution by the computing devices coupled thereto, so that at least first and second computing devices among the plurality execute different first and second test programs from the suite substantially simultaneously. The server is further adapted to receive  
10 messages via the communication interface from the computing devices with respect to execution of the test programs, the messages containing the respective unique identifier, and to control the execution of the test programs in the suite based on the messages and the respective unique identifier therein by  
15 communicating responses to the messages via the communication interface, wherein each of the responses is addressed to a respective one of the computing devices that is associated with the respective unique identifier.

[0017] According to an aspect of the server, the computing devices are coupled to the communication interface via a  
20 common test host, wherein an identifier of the common test host is shared by each of the computing devices, and the identifier of the common test host is included in the respective unique identifier thereof.

## 25 BRIEF DESCRIPTION OF THE DRAWINGS

[0018] For a better understanding of the present invention, reference is made to the detailed description of the invention, by way of example, which is to be read in conjunction with the following drawings, wherein like elements are given  
30 like reference numerals, and wherein:

[0019] Fig. 1 is a block diagram that schematically illustrate systems for parallel testing of low-end computing de-

vices, in accordance with an embodiment of the present invention;

[0020] Fig. 2 is a block diagram that schematically illustrate systems for parallel testing of low-end computing devices, in accordance with an alternate embodiment of the present invention;

[0021] Fig. 3 is a block diagram that schematically illustrates program components used in a test system, in accordance with an embodiment of the present invention;

[0022] Fig. 4 is a detailed flow chart that schematically illustrates a method for parallel testing of low-end computing devices, in accordance with an embodiment of the present invention; and

[0023] Fig. 5 is a flow chart that schematically illustrates a method for parallel testing of low-end computing devices, in accordance with an embodiment of the present invention.

#### **DETAILED DESCRIPTION OF THE INVENTION**

[0024] In the following description, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent to one skilled in the art, however, that the present invention may be practiced without these specific details. In other instances well-known circuits, control logic, and the details of computer program instructions for conventional algorithms and processes have not been shown in detail in order not to unnecessarily obscure the present invention.

[0025] Software programming code, which embodies aspects of the present invention, is typically maintained in permanent storage, such as a computer readable medium. In a client/server environment, such software programming code may be stored on a client or a server. The software programming code



may be embodied on any of a variety of known media for use with a data processing system, This includes, but is not limited to, magnetic and optical storage devices such as disk drives, magnetic tape, compact discs (CD's), digital video discs (DVD's),  
5 and computer instruction signals embodied in a transmission medium with or without a carrier wave upon which the signals are modulated. For example, the transmission medium may include a communications network, such as the Internet.

[0026] Reference is now made to Fig. 1, which is a  
10 block diagram that schematically illustrates a system 20 for parallel testing of multiple mobile information devices 24, in accordance with an embodiment of the present invention. The system 20 is built around a test server 22, which is described in greater detail hereinbelow. The devices 24 are client de-  
15 vices, and are typically low-end devices, with limited computing power and memory, for example, cellular telephones or personal digital assistants (PDA's). In the description that follows, the devices 24 are assumed to comply with MIDP, but the principles of the present invention are equally applicable to  
20 other types of low-end computing devices, operating in accordance with other standards and specifications. The server 22 typically comprises a programmable processor, and has suitable communication interfaces, such as wireless or wired interfaces, for communicating with multiple devices 24 simultaneously.

[0027] Each of the devices 24 receives a unique identifier for communicating with the server 22. Typically, the unique identifier may comprise a unique Internet Protocol (IP) address that is assigned to each of the devices 24 for communicating with the server 22. Alternatively, the server may assign  
25 IDs of other types, or the ID's may be assigned by a user upon initiating communication between one or more of the devices 24 and the server 22. Methods for assigning and using these IDs are described in detail hereinbelow.

[0028] Reference is now made to Fig. 2, which is a block diagram that schematically illustrates a system 30 for parallel testing of multiple devices 24, in accordance with another embodiment of the present invention. In this embodiment, the server 22 communicates with the devices 24 through a test host 32, such as a personal computer or workstation. Multiple test hosts of this sort may be connected to the server 22 in parallel, but only a single host is shown in Fig. 2 for the sake of simplicity. The host 32 can simultaneously accommodate multiple devices 24, but the host 32 typically has only a single IP address. Therefore, in this embodiment, the IP address cannot be used conveniently to identify the individual devices 24, and an alternative unique identifier is typically used, as described below.

[0029] Reference is now made to Fig. 3, which is a block diagram that schematically illustrates software program components running on the server 22 and the devices 24, in accordance with an embodiment of the present invention. Elements of this software may be provided to the server 22 and to the devices 24 on tangible media, such as optical or magnetic storage media or semiconductor memory chips. The software may be downloaded to the server 22, and alternatively or additionally, to the devices 24 in electronic form, for example, over a network or over the air.

[0030] The server 22 comprises a test framework 40, which generates and deploys the tests to be carried out by the devices 24. The test framework 40 may be implemented as the "Java Device Test Suite" execution framework (JDTS) (version 1.0 or higher), available from Sun Microsystems, Inc., which employs MIDP. A suitable version of the test framework 40 is described, for example, in the above-mentioned Application No. (STC File No. 47900), which is commonly assigned herewith, and is herein incorporated by reference.

[0031] The tests typically are packaged in the form of Java applications contained in a set of JAD and JAR files. Each JAR file of this sort, together with its accompanying JAD file, is referred to hereinbelow as a test bundle 52. Users of the system 20 (Fig. 1) or the system 30 (Fig. 2) interact with the test framework 40 in order to select the tests to be executed by the system. Alternatively, other test frameworks may be used for generating the required test files, as will be apparent to those skilled in the art.

[0032] A test manager 42 in the server 22 is responsible for serving requests from the devices 24, based on the unique client identifiers mentioned above. Typically, whenever one of the devices 24 makes a request, the test manager 42, typically operating as a main thread, reads the request and assigns a new thread 44 to handle it. This thread 44 retrieves the client unique identifier from the request, calls the components of the test framework 40 that are needed to process the request, and then returns the appropriate response to the client device, as described hereinbelow. After assigning the thread 44 to handle the client, the main thread of the test manager 42 waits for the next client request. Each client request is handled by a separate thread 44, which terminates upon completion of processing. This arrangement, together with the unique identifier mechanism, ensures that the server 22 will be able to handle multiple devices 24 simultaneously without confusion.

[0033] In order to run Java applications, the devices 24 contain an implementation of the Connected Limited Device Configuration specification, CLDC 46, with an implementation of the Mobile Information Device Profile specification, MIDP 48, running over the CLDC 46. The applications that run on this technology, such as the tests supplied by framework 40, are known as MIDlets. These applications are created by extend-

ing an API MIDlet class of the MIDP 48. Thus, each test bundle 52 is actually a MIDlet, packaged in the form of a JAD/JAR file pair.

**[0034]** The test bundle 52 is typically downloaded to the devices 24 in a two-step process:

**[0035]** 1. The server 22 downloads the JAD file, which contains environment settings and some environment demands. Application Manager Software, AMS 50, which is typically a part of a browser built into the devices 24, evaluates the JAD file to ensure that the device is able to accept the MIDlet. For example, the JAD file for a given MIDlet may specify that the device must support MIDP version 2.0. If the device does not support this version, the AMS 50 rejects the application download, and saves the time that would otherwise be consumed by downloading the much larger JAR file.

**[0036]** 2. After completing all the relevant checks, the AMS 50 reads from the JAD file the location of the corresponding JAR file on the server 22 and asks to download the JAR file to one or more of the devices 24. The JAR file contains all the relevant classes of the test bundle 52.

**[0037]** Once the JAR file for the test bundle 52 is downloaded to one of the devices 24 and stored in the local device memory, the device is ready to run the tests of the test bundle 52. Every JAR file that the AMS 50 downloads to the devices 24 typically contains an agent 54, which is used to run the tests, in addition to classes corresponding to the tests themselves. To start test execution the AMS 50 runs the agent class. The agent 54 then addresses the server 22 in order to receive instructions regarding the next test to run (getNextTest) and to report test results (sendTestResult),

typically using a protocol based on HTTP. Each test in the test bundle 52 corresponds to a respective class in the JAR file. Each client request that is addressed by the agent 54 to the server 22 includes the unique identifier that has been assigned to the particular one of the devices 24, so that the server 22 is able to recognize the client and serve it in the correct manner.

#### **Implementation Details.**

[0038] Further details of the implementation of the server 22 are given in Listing 1 (class BaseHttpServer). An implementation of the communications interface through which requests and messages are transmitted between the server 22 and the devices 24 is detailed in Listing 2 (class Communicator). Runtime generation of JAD files by the server 22 is accomplished using Listing 3 (class HttpServer). Launching of the agent 54 is detailed in Listing 4 (class MIDPRunner). Implementation of the thread 44 is detailed in Listing 5 (class Server-TaskThread).

[0039] Listing 6 shows a class (class Extender) that is needed by the classes shown in Listings 1 - 5. A brief description of Listing 6 follows.

[0040] A public interface Extender provides access to a class Extender. The class Extender enables an agent link with platforms that require extensions of their main application class, for example to properly employ a system class, such as class Applet or class MIDlet. The class Extender accepts delegation of platform specific commands from an agent.

[0041] The interface Extender includes the following methods. A method getRunnerExtender retrieves a reference to a platform class, which the main application class extends. Using this method, an agent provides access to the test program by the main application class in the context in which it is cur-

rently executing. An object is returned, which can be cast to the system class that the extender class extends. A method terminateAgent provides a platform-specific way of application termination.

5       **[0042]**       It will be understood that Listings 1 - 6 are exemplary, and the functions and operations shown therein can be accomplished using other techniques known to the art.

10       **[0043]**       Reference is now made to Fig. 5, which is a high level flow chart that schematically illustrates a method for running test suites on multiple client devices 24 in the system 20 (Fig. 1) or the system 30 (Fig. 2), in accordance with an embodiment of the present invention. The flow chart in Fig. 5 presents an interaction involving only a single client request for clarity. However, the method can be performed simultaneously, with many clients. Indeed, different devices may be executing different tests, or even different test suites or test bundles at any given time. This method is explained with reference to the software structures shown in Fig. 3, although other implementations are also possible, as will be apparent to those skilled in the art. The method begins at initial step 100, which is a configuration step. A server is attached to a plurality of client devices to be tested using suitable communications links.

25       **[0044]**       Next, at delay step 102 the server awaits a request from a client. As will be apparent from the discussion below, the request could be for a new test bundle, or for the next test in a test bundle that is currently executing.

30       **[0045]**       Upon receipt of a client request, control proceeds to decision step 104. Here it is determined whether the client request received at delay step 102 is a request for a new test bundle. This is normally the case when the client is first recognized by the server. Otherwise, such a request can

occur if a previous test bundle has been completed by a client already known to the server according to its unique identifier.

**[0046]** If the determination at decision step 104 is negative, then generally, the server is already aware of the requesting client. Control proceeds to decision step 106, which is disclosed below.

**[0047]** If the determination at decision step 104 is affirmative, it is concluded that the server has not previously interacted with the requesting client. Control proceeds to step 108. Here a unique identifier is assigned to the requesting client. Whichever of the alternate methods disclosed herein for making the assignment is employed, the client is uniquely identified at step 108, and its subsequent requests and results will be handled without possibility of confusion with other currently attached clients. As noted above different clients may be identically configured, and may even be concurrently executing the same test bundle. Furthermore, any test results reported by the now uniquely identified client are accurately associated with that particular client so as to guarantee the integrity of test reports that may be eventually generated by the server. Control now proceeds to step 110.

**[0048]** At step 110 a JAD file corresponding to the client request is generated or selected by the server for transmission to the client. Control then proceeds to step 112, which is disclosed below.

**[0049]** Decision step 106 is performed when the determination at decision step 104 is negative. Here it is determined if the client request received at delay step 102 is a request for the next test to be executed in a current test bundle. Such requests may be generated at the client responsively to evaluation at the client of a previously downloaded JAD file, based on suitability of the tests for the particular client.

[0050] If the determination at decision step 106 is affirmative, then control proceeds to step 114. The server retrieves the test record that corresponds to the next test to be executed by the client. It will be apparent that this mechanism provides a high degree of central control by the server, so as to optimize the order of test execution by different clients. For example, if the server has received borderline test results from the client, it could elect to repeat a particular test, or to perform supplemental tests that would otherwise be skipped.

[0051] If the determination at decision step 106 is negative, then it is concluded that an unrelated client request has been made. For example, the client may have requested transmission of test results, or a display illustrating the profile of a test. Control proceeds to step 116, where this request is processed.

[0052] Next, at step 112 a response to the client request is assembled. Test information obtained in step 114, a JAD file obtained in step 110, or information relating to the request processed in step 116, whichever is applicable, is now concatenated with the client's unique identifier.

[0053] Next, at step 118, the response assembled at step 112 is downloaded to the requesting client. Control then returns to delay step 102, where the next client request is awaited.

[0054] Reference is now made to Fig. 4, which is a detailed flow chart that schematically illustrates the method shown in Fig. 5 in further detail, in accordance with an embodiment of the present invention. The method begins with selection of the tests to be run, at a test selection step 60. This step is generally performed by a user, such as a development engineer, through interaction with the framework 40. Based on the user selections, the framework 40 deploys the selected tests, at a deployment step 62. At this step, the test frame-



work creates a list of test bundles 52 (JAD/JAR file pairs), which also include the agent 54, as described above. When the deployment phase is completed, bundles 52 are ready to be downloaded to the devices 24, and the server 22 waits for the devices 24 to connect at a server waiting step 64.

**[0055]** Each of the devices 24 that is linked to the server 22 makes an initial connection with the server 22 and requests a test bundle 52, at a bundle request step 66. The syntax for this initial request is typically:

`http://<server_name>:<server_port>/getNextApp.jad.`

**[0056]** Every time one of the devices 24 addresses the server 22 at step 66, the server assigns the device a unique identifier. The server 22 then sends a JAD file to the client device containing the unique identifier, along with other information regarding the test bundle 52, at a JAD download step 68. Typically, the server 22 marks the associated JAR file as "in use," to ensure that the same test bundle is not inadvertently assigned to two devices 24. The AMS 50 stores the unique identifier in the local memory of the appropriate one of the devices 24. Thereafter, each time this device addresses the server, it retrieves the unique identifier from its memory and concatenates the unique identifier to the request in order to request the next test to perform, for example as follows:

`http://<server_name>:<server_port>/getNextTest/<ID>.`

**[0057]** If each of the devices 24 that is linked to the server 22 has a unique IP address, this IP address may be used by the server 22 as the unique identifier for the respective device. When the devices 24 communicate with the server 22 using HTTP, one of the client request parameters is simply the IP source address, so that the unique identifier is naturally contained in every request.

**[0058]** Alternatively, in some cases, such as the system 30 (Fig. 2), multiple devices 24 may use the same IP ad-

dress in communicating with the server 22. Thus, upon receiving the initial client request at step 66, the server 22 may recognize that the IP source address is already in use as a unique identifier by another one of the devices 24. In this case, the test manager 42 creates a new unique identifier for the current client device, typically by concatenating the client's IP address with a sequential number, to which the server 22 has access, and which is inaccessible to network elements other than the server 22 and the devices 24.

**[0059]** Further alternatively, an outside ID manager (not shown) may be used to assign client unique identifiers, either automatically or under control of the user. Thus, for example, if the initial connection request issued by one of the devices 24 has the form:

http://<server\_name>:<server\_port>/getNextApp,  
the request is extended by the ID manager to be  
http://<server\_name>:<server\_port>/getNextApp/1. The next connection request, by another client device, is extended to be  
http://<server\_name>:<server\_port>/getNextApp/2; and so on. The  
server 22 assumes that the outside ID manager is reliable, and assigns to each of the devices 24 a respective unique identifier that is appended to the first request from the device.

**[0060]** In deciding which JAD file to send at step 68, the server 22 consults a list of test bundles created at step 62 to determine which test bundles have not yet been assigned. It may parcel out the different test bundles among different devices 24 in such a way that the testing load is balanced among the devices, and all the devices 24 therefore complete their respective shares of the test suite at approximately the same time. The load balancing is done according to a controlling policy, which is not necessarily according to execution time of the different test bundles or components of the test bundles. The test manager 42 may also take into account

sequential testing relationships among the test bundles, for example, that a test bundle B should be carried out only by a client device that has successfully passed the tests in a test bundle A. On the other hand, the server 22 may decide at step 68 not to send any test bundle to the client device in question, for example because there are no more test bundles to execute. If a client device does not receive a JAD file after submitting its request at step 66, the device exits from the test program, at a client termination step 70.

10       **[0061]**       Assuming one of the devices 24 receives a JAD file, however, the AMS 50 checks the environment settings in the JAD file, at an environment checking step 72. The AMS may determine that the MIDP 48 or the CLDC 46 or other resources of the device, such as memory or display capacity, are incompatible with the environment settings required for the test bundle 52, at an environment rejection step 74. In this case, as well, the client device exits, after notifying the user of system 20 that the settings are incorrect, and test execution stops.

20       **[0062]**       Once the AMS 50 has determined that one of the devices 24 is able to carry out the test bundle indicated by the JAD file, it asks the test manager 42 to download the corresponding JAR file, at a JAR request step 76. The location of the JAR file on the server 22 is provided by the JAD file, and this location is invoked by the AMS 50 in requesting the JAR file. The test manager 42 reads the JAR file from the test framework 40 and downloads it to the device at a JAR download step 78. The AMS 50 stores the JAR file in the local memory of device and runs the class of the agent 54 to begin the tests in the test bundle 52.

30       **[0063]**       When the agent 54 is invoked in this manner, it retrieves the unique identifier of one of the devices 24 from the local memory of the device, at a first unique identifier

retrieval step 80. It will be recalled that the unique identifier was passed from the server 22 to the device in the JAD file downloaded at step 68. The agent 54 uses this unique identifier in asking the server 22 for the name of the next test to be run, at a next test request step 82. This request has the general form:

http://<server\_name>:<server\_port>/getNextTest/<ID>.

The server 22 uses the unique identifier to determine the next test to be run in the bundle currently assigned to this client device. The server 22 returns the name of the next test - actually the class name of the desired test in the current test bundle 52 - to the client device, at a next test determination step 84.

**[0064]** Upon receiving the reply from the server 22, the agent 54 ascertains that the reply has named one of the classes in the present test bundle, at a next test checking step 86. If so, the agent runs the class named by the server 22 at a test execution step 88. Upon completing the test corresponding to the named class, the agent 54 prepares to report the test results to the server 22. For this purpose, the agent 54 again reads the unique identifier of the device, at a second unique identifier retrieval step 90. It uses this unique identifier in reporting the test results to the server 22, at a result reporting step 92, in the general form:

http:// <server\_name>:<server\_port>/sendTestResults/<ID>.

The server 22 receives the test record from the device, and adds the record to a test report, at a test recording step 94. This report is later submitted to the user of the test system upon completion of the test suite, or upon demand by the user. The agent 54 then returns to step 82 to request the next test to run.

**[0065]** On the other hand, the server 22 may determine at step 84 that there are no more tests to run in the current

test bundle. In this case, at step 86, the agent 54 is informed that the test bundle has been completed. The agent 54 returns control to the AMS 50, which then asks the server 22 for the next bundle of tests to be executed, at step 66. This process continues until the entire test suite specified at step 60 is completed, unless the server 22 exits earlier due to a system error.

[0066] Although the embodiments described hereinabove are based on the Java programming language and Java-related conventions, as well as on certain specific protocols and device specifications, such as CLDC and MIDP, the principles of the present invention may similarly be applied to low-end computing devices using other languages, conventions, protocols and specifications. It will thus be appreciated that the embodiments described above are cited by way of example, and that the present invention is not limited to what has been particularly shown and described hereinabove. Rather, the scope of the present invention includes both combinations and subcombinations of the various features described hereinabove, as well as variations and modifications thereof, which would occur to persons skilled in the art upon reading the foregoing description and which are not disclosed in the prior art.

#### COMPUTER PROGRAM LISTINGS

##### Listing 1

```
public abstract class BaseHttpServer implements Runnable {  
    /**  
    * "Runnable" is a standard Java class.  
    *  
    * Current tests is a hash table, which hold current tests  
    * for each midp client  
    * according to its signature.  
    */
```

```
private Hashtable currentTests = new Hashtable ();

/**
5  *The class that manage the distribution of the
  * bundles to the clients,
  * and dealing with requests that related to the
  * bundles such as get next test.
  */
10 public TestProvider getTestProvider() {
    return testProvider;
}

/**
15 *Set the current test of a specific client.
  */
    public void setCurrentTest
        (String signature, byte[] args)
20 {
    currentTests.put (signature, args);
}

/**
25 *Get the current test of a specific client.
  */
    public byte[] getCurrentTest(String signature)
    * {
        byte[] currentTest =
    *     (byte[]) currentTests.get (signature);
30     return currentTest;
    }

/**
35 *Remove the current test of a specific client.
  */
    public void removeCurrentTest(String signature){
        currentTests.remove (signature);
    }

40
    public void setTestProvider(TestProvider tp) {
        testProvider = tp;
        mainClass = tp.getAppMainClass();
        // The MIDlet name
45
        if (mainClass == null) {
            throw new NullPointerException
                ("Provider main class not defined");
        }
    }
}
```

```
jarSourceDir = tp.getJarSourceDirectory();
    //the directory of the JAR files.
    if (mainClass == null) {
        throw new NullPointerException
5         ("Provider jar source directory not defined");
    }
}

10
public void run() {
    Socket conn = null;
    aborted = false;
    started = true;
15    synchronized(this) {
        notify();
    }

    while (!done) {
20        try {
            try {
                conn = socket.accept();
                //retrieve the IP from the connection
                String ipAddress =
25                conn.getInetAddress().getHostAddress();

                //assign new thread to process
                // the client request.
                ServerTaskThread task =
30                new ServerTaskThread(ipAddress, conn, this);

                //process request data
                task.processRequest ();

                //start that task.
35                task.start ();
            }
            catch (InterruptedException iie) {
                continue;
40            }
        }
        catch (Exception x) {
            x.printStackTrace();
        }
45    }

    synchronized (socket) {
        try{
            aborted = true;
```

23

```
        LOCALHOST = false;
        nextID = 1;
        socket.close ();
        socket.notifyAll();
5      }
      catch (IOException ioe){
        ioe.printStackTrace ();
      }
10    }
  }

15  /** reads alternative JAM tags if "jam.tags"
   * property file provided
   * Keys:
   * AppURL - Application URL JAM Tag
   * AppSize - Application Size JAM Tag
20  * MainClass - Application Main Class JAM Tag
   * AppName - Application Name JAM Tag
   * AppUseOnce - Use-Once JAM Tag
   */
   public abstract void loadTagNames();
25

   /** Runtime generation of JAD file, to be sent
   * to the client as result of
   * getNextApp request.
30  */
   public abstract ByteArrayOutputStream
     generateJAM (String host,
                 int port,
                 String next_app,
35                 long length,
                 String mainClass,
                 String jarLocation,
                 String protocolPermissions,
                 String agentMonitorID,
40                 String signature,
                 // attaching the client ID to the JAD file.
                 ServerTaskThread task)
                 throws UnsupportedOperationException;

45  /**
   * In case that a certain ID is already in use,
   * than the server assign an ID to the client.
   */
```



```
public int getNextID(){
    return nextID++;
}

5  /**
   * When an assigned thread was completed its work,
   * we need to remove it from the list of threads
   * currently running in the system.
   */
10 public void cleanUp(String signature){
    synchronized(taskHashtable){
        taskHashtable.remove (signature);
    }
15
    /**
    * When this class assign new thread for processing
    * the client request, it puts
    * the thread in a hashtable for keeping reference
20 * to it. If a former thread,
    * that serve a particular client, has not completed
    * yet, the former thread is
    * terminated (generally this case happens
    * when something went wrong).
25 */
    public void putTaskThread (String signature,
        ServerTaskThread serverTaskThread){
        synchronized(taskHashtable){
            ServerTaskThread task =
30             (ServerTaskThread)getTaskThread(signature);
            if (task != null){
                verboseIn("task is being stoped");
                task.stopTask();
                task = null;
35             }
            taskHashtable.put(signature,
                serverTaskThread);
        }
40
    /**
    *Get the reference to a client's thread
    * that currently serve the client request.
    */
45 public ServerTaskThread
    getTaskThread ( String signature){
        synchronized(taskHashtable){
            return (ServerTaskThread)
                taskHashtable.get (signature);
        }
    }
}
```

```

    }
}

```

5

## Listing 2

```

/**
 * Communicator is an implementation of client-server
10  * communication between a test loader
 * and the main JDTS application. The client uses
 * this class in order to make the requests
 * to the server. When the request is ready to be sent,
 * the communicator attaches the client ID
15  * to the request and then send it to a Sender class
 * that communicates with the server using HTTP.
 */

public class Communicator{
20

    /** Communicates with the main JDTS application test
    provider. Retrieves current executing test.
    * @return Test object
25  */
    public Test getCurrentTest() {
        return getTest("/getCurrentTest/");
    }

    /** Communicates with the main JDTS application test
    provider. Retrieves next test to execute.
    * @return Test object
    */
30  public Test getNextTest() {
        return getTest("/getNextTest/");
35  }

    /** Get the test based on the command specified
    * @return Test object
    */
40  protected Test getTest(String command) {
        try {
            /**
45  * client sends get next test command to the server,
            * through the client class,
            * using the getSignature() function, that retrieves the
            * ID from the environment.

```

```
    */
    byte[] bytes = client.getNextTest(command +
        getSignature());
5      .
      .
      .
10  }

    /** Reports results of the test execution
     * @param Test object, which should include test log
     * and execution status (test result)
15  */
    public void reportTestResults(Test currentTest) {
        //send test results as byte[] using the client ID.
        client.sendTestResult("/sendTestResult/" +
            getSignature(), encoder.getBytes());
20  }

    /**
25  * Retrieve the the ID from the JAD file. When calling
     * getNextApp, the server attached to the JAD file the ID
     * that comes with
     * the request. If there is no ID attached, the ID is
     * the IP address of the agent.
     * This method returns empty String in case that there is
30  * not ID attached to the JAD file.
     * Any Communicator that extends this class,
     * probably overrides this function to supply
     * its own way for getting the signature from the platform
     * environment.
35  */

    public String getSignature() {
        MIDlet midlet = (MIDlet)agentManager.getExtender();
        String signature =
40      midlet.getAppProperty("Bundle-Signature");

        if (signature == null){
            return "";
        }
45      else{
            return signature;
        }
    }
}
```

```
/** Request to perform a communication action in addition
 * to the actions explicitly defined in the interface.
 * @param actionName is the name of the action to be
5  * performed. The implementation filters actions according
 * to action name.
 * @param actionObjects is the array of objects this
 * action might use. The exact type of this objects needs
10 * to be known
 * to both class who calls the method and to the
 * Communicator implementation
 */

public Object performAction(String actionName,]
15  Object[] actionObjects) {

    Object[] obj = new Object[1];

    if (actionName.equals("ShowTestDescription")){
20         return showTestDescription((String)
            actionObjects[0]);
    }

    else if (actionName.equals
25  ("ShowTestDescriptionString")){
        obj[0] = actionObjects[0];
        return showTestDescriptionString
    }
```

```
(obj, (String)actionObjects[1]);
}

else if (actionName.equals
5   ("ShowCombineTestDescription")){
    obj[0] = actionObjects[1];
    return ShowCombineTestDescription
        ((String)actionObjects[0], obj);
}

10 else if (actionName.equals
    ("RecordedShowCombineTestDescription")){
    obj[0] = actionObjects[1];
    return RecordedShowCombineTestDescription
15    ((String)actionObjects[0], obj);
}

else
    return null; //unknown action
20 }

/**
 * This set of functions create the desired requests
25 * to be sent to the JDTS server.
 * The client ID is attached to each request.
 */
private String ShowCombineTestDescription
    (String descFileAndButtons, Object[] outputString) {
30
    System.out.println
        ("Communicator.ShowCombineTestDescription:" +
            descFileAndButtons);

35    byte[] bytes = client.sendCommand
        ("/ShowCombineTestDescription/" + descFileAndButtons +
            "/" + getSignature(), outputString);

    if (bytes != null){
40        String testResult = new String(bytes);
        return testResult;
    }
    else
        return null;
45 }

private String RecordedShowCombineTestDescription
    (String descFileAndButtons, Object[] outputString) {
```

```
System.out.println
    ("Communicator.RecordedShowCombineTestDescription:" +
     descFileAndButtons);

5   byte[] bytes = client.sendCommand
    ("/RecordedShowCombineTestDescription/" +
     descFileAndButtons + "/" + getSignature(),
     outputString);

10  if (bytes != null){
        String testResult = new String(bytes);
        return testResult;
    }
    else
15     return null;
}

/** Implementation of show test description request
20 */
private String showTestDescription(String description) {

    System.out.println
        ("Communicator.showTestDescription:" + description);

25   byte[] bytes = client.sendCommand("/showTestDescription/"
        + description + "/" + getSignature(), null);

    if (bytes != null){
30         String testResult = new String(bytes);
        return testResult;
    }
    else
        return null;
35 }

private String showTestDescriptionString
    (Object[] testInfo, String buttonsArray) {
40   byte[] bytes = client.sendCommand
    ("/showTestDescriptionString/" + buttonsArray + "/" +
     getSignature(), testInfo);

    if (bytes != null){
45         String testResult = new String(bytes);
        return testResult;
    }
    else
        return null;
```

}

}

5

## Listing 3

```

/**
 * Implements the creation of the JAD file.
 */
10 public class HttpServer extends BaseHttpServer {

    /**
     * BaseHttpServer extender applicable for MIDP project
     */
15     //JAM TAGS (default values set)
     private String profile = "MicroEdition-Profile";
     private String configuration = "MicroEdition-
Configuration";
20     private String appURL = "MIDlet-Jar-URL";
     private String appSize = "MIDlet-Jar-Size";
     private String appVersion = "MIDlet-Version";
     private String appName = "MIDlet-Name";
     private String appVendor = "MIDlet-Vendor";
25     private String midlet = "MIDlet-1";
     private String useOnce = "Use-Once";
     private String bundleSignature = "Bundle-Signature";
     private String permission = "MIDlet-Permissions";

30

     public HttpServer() {
         super();
         ALTERNATIVE_NEXT_APP = "getNextApp.jad";
35         loadTagNames();
     }

40

     /** Runtime generation of JAM file (applicable for
     DoJa project)
     */
45     public synchronized ByteArrayOutputStream
generateJAM(String host,
        int port,
        String next_app,

```

```

        long length,
        String mainClass,
        String jarLocation,
        String protocolPermissions,
5       String agentMonitorID,
        String signature,
        ServerTaskThread task) throws
            UnsupportedEncodingException {

10     ByteArrayOutputStream bos = new ByteArrayOutputStream();
        PrintWriter pw = new PrintWriter(new
        OutputStreamWriter(bos, "ISO8859_1"));

        File jar_path = new File(jarLocation, next_app);
15     long jarsize = jar_path.length();

        pw.println(appURL + ": http://"
            + host
            + ":" + port + "/" + next_app);
20     pw.println(appSize + ": " + jarsize);
        pw.println(configuration + ": CLDC-1.0");
        pw.println(profile + ": MIDP-1.0");
        pw.println(appName + ": TestSuite"); // needed by JAM
        pw.println(appVersion + ": 1.0");
25     pw.println(appVendor + ": Sun Microsystems, Inc.");
        pw.println(midlet + ": TestSuite,, " + mainClass);
        pw.println(useOnce + ": yes");
        pw.println(permission + ": " + protocolPermissions);
        //the unique ID is sent to the client.
30     pw.println(bundleSignature + ": " + agentMonitorID);

        //add meta-info: jad properties specified for the test
        try {
            Hashtable metaInfo =
35     getTestProvider().getCurrentAppInfo(signature);
            String[] jadprops = (String[])
            metaInfo.get("JadParameters");
            if (jadprops != null) {
                //write these props to the jad
40         for (int i = 0; i < jadprops.length; i++)
                pw.println(jadprops[i]);
            }
            } catch (Exception e) {
                e.printStackTrace();
45         System.out.println("Failed to add jad parameter.
        [IGNORED]");
            }

        pw.close();

```



```

        task.setResponseProperty("Content-Type",
        "text/vnd.sun.j2me.app-descriptor");
        task.setResponseProperty("Content-Length", "" +
5    bos.size());

        return bos;
    }

10 }

```

## Listing 4

```

/**
15  * Agent launcher extender compatible with MIDlet.
    * MIDPRunner is an agent launcher for the MID
    * Profile platform.
    * When the AMS completes the download
    * of the application (JAR file),
20  * it addresses the JAD file to get the class name
    * (the MIDlet name) to be executed.
    */

public class MIDPRunner extends MIDlet implements Extender
25 {

    private CLDCRunner runner = null;
    private String bundleID = null;

30    public void startApp() {
        /**
        * this is the way get the client ID from the enviroment
        * (from the JAD file)
        */
35        bundleID = getAppProperty("BundleID");

        .
        .
        .
    }

40    public void pauseApp(){}
    public void destroyApp(boolean unconditional) {}

    /**
45  * Returns MIDPRunner extender object as a MIDlet.
    * In order to get environment settings from other class,
    * you need to get access to this MIDlet.
    */

```

33

```

public Object getRunnerExtender() {
    return (MIDlet) this;
}

5.    /** Provides MIDP specific way to terminate agent
        */
    public void terminateAgent() {
        notifyDestroyed();
    }
10   }

```

## Listing 5

```

/**
15  * This class is the implementation of the
    * thread 44 (Fig. 3). It is used by the server
    * to carry out the client request tasks.
    * According to the client request, this thread
    * communicates with the corresponding
20  * JDTS component and supplies the client request.
    * Once the thread is started, the main thread of the
    * server returns to listen to new requests from other
    * clients, and the thread handles communication with
    * the current client, until the current request has
25  * been fulfilled.
    */

public class ServerTaskThread extends Thread {
    /**
30    * Default get next app.
    * "Thread" is a standard Java class.
    */

    protected final String NEXT_APP = "getNextApp";
35    /** Alternative NEXT_APP STRING. Can be adapted per
        * platform
        */
    protected String ALTERNATIVE_NEXT_APP = "getNextApp";
    protected final String NEXT_TEST = "getNextTest";
40    protected String NEXT_CMD = "/getNextCommand";
    protected String CURRENT_TEST = "getCurrentTest";

    /**
45    * Sets some variable for later execution.
    */
    public ServerTaskThread(String ipAddress, Socket conn,
        BaseHttpServer httpServer) {

```

```

        this.ipAddress = ipAddress;
        jarSourceDir = httpServer.getJarSourceDir();
        mainClass = httpServer.getMainClass();
    }
5
    /**
    * This function sets the initial variable, and prepares
    * it to serve the client request.
    * The thread 44 parses the client request, retrieves the
10  * client ID, and checks at the main server
    * if it is to be serve a client-specific request.
    */
    public void processRequest(){
        String line, method, url, version;
15        StringTokenizer cutter;
        try{
            in = new
                DataInputStream(conn.getInputStream());
            raw = conn.getOutputStream();
20            out = new PrintWriter(new
                OutputStreamWriter(raw, "ISO8859_1"));

            if (!httpServer.getBatchMode()){
                NotifierDialog.disposeNotification();
25            }

            line = in.readLine();

            if (line == null) {
30                verboseIn
                    ("Connection failure. Retrying.");
                return;
            }

            verboseIn("got new request: " + line);
            if (line.indexOf("getNextApp") != -1){
                getNextAppReq = true;
            }
            else {
40                getNextAppReq = false;
            }

            cutter = new StringTokenizer(line);
            method = cutter.nextToken();
45            url = cutter.nextToken();
            version = cutter.nextToken();

            readHeaders(in);

```

35

```

    if (method != null && url != null && version
        != null) {
        try {
            try {
5              request = new URL(url);
            } catch (MalformedURLException e) {
                request = new URL("http", host,
                    port, url);
            }
10          //set the client signature.
            signature = setSignature(ipAddress,
                request.toString());

            if (method.equals(GET)) {
15              taskName = HANDLE_GET;

                httpServer.putTaskThread
                    (signature, this);

20              } else if (method.equals(POST)) {
                taskName = HANDLE_POST;

                httpServer.putTaskThread(signature,
                    this);
25              } else {
                verboseIn
                    ("unknown request method: " +
                        method);
                sendDiagnostics
30                  (HTTP_BAD_METHOD, out);
            }
            } catch (MalformedURLException mue) {
                System.out.println
                    ("Http server error: " + mue);
35                sendDiagnostics
                    (HTTP_BAD_REQUEST, out);
            }
        }
    } catch (IOException ioe){
40        ioe.printStackTrace();
    }
}

/**
45 * The main server class, after assurance
 * that all the parameters for the request
 * were set, calls this function
 * to start processing the request.
 */

```

```

public void run(){
    try{
        if (taskName.equals(HANDLE_GET)){
            handleGet(request, out, raw, in);
5           }
            else if (taskName.equals(HANDLE_POST)){
                handlePost(request, in, out, raw);
            }
            else{
10             System.out.println("Http server error");
                sendDiagnostics(HTTP_BAD_REQUEST, out);
            }
        }catch (Exception e){
            e.printStackTrace();
15        }
        cleanUp();
    }

20    /**
    * This function processing HTTP GET requests.
    * The request is retrieved from the HTTP request line.
    * Then, according to the type of command
25    * the coresponding functionality is being used.
    */
    protected void handleGet(URL request, PrintWriter out,
        OutputStream raw, DataInputStream in) throws
        IOException, Exception {
30
        /**
        *there is no need to retrieve content from the
        * client, because this method handles
        * GET requests.
35    */
        String ref = (request.getRef() == null) ? ""
            : "#" + request.getRef();
        String path = request.getFile() + ref;

40        if (path.startsWith(testRoot + NEXT_APP)
            || path.startsWith(testRoot +
                ALTERNATIVE_NEXT_APP)) {

            verboseln(path);

45            String next_app = null;
            try {
                //request for next bundle.
                next_app =

```

37

```
        testProvider.getNextApp(signature);
    }
    catch (RuntimeException e) {
        e.printStackTrace();
5      System.out.println
      ("Unable to provide next bundle possibly due to a problem
      on the Client side");
        System.out.println
10      ("Client notified to stop the execution");
        sendDiagnostics(HTTP_NOT_FOUND, out);
      //notification to Client impl that no more bundles
      available.
        return;
    }
15
    if (next_app == null) {
        sendDiagnostics(HTTP_NOT_FOUND, out);
      //notification to Client impl that no more bundles
      //available.
20      return;
    }

    if (next_app.length() == 0) {
      //notification to Client impl that bundles
25      // currently unavailable.
      //Client should wait and retry again
      sendDiagnostics(HTTP_UNAVAILABLE, out);
      return;
    }
30

    //check tags
    httpServer.checkTags();

    verboseIn("next_app: " + next_app);
35    File jar_path = new File(jarSourceDir,
        next_app);
    long length = jar_path.length();

    String agentMonitorID =
40      getAgentMonitorID(request.toString());

    //call for creating JAD file, and downloaded it
    ByteArrayOutputStream bos =
45      httpServer.generateJAM(host,
        port, next_app, length,
        mainClass, jarSourceDir,
        protocolPermissions, agentMonitorID,
        signature, this);
```

38

```

        sendDiagnostics(HTTP_OK, out);
        raw.write(bos.toByteArray(), 0,
            bos.size());
        raw.flush();
5      }
      // cmd "get next test"
      else if (path.startsWith(testRoot + NEXT_TEST)) {
        verboseIn(NEXT_TEST);
10      //get the next test from the bundle that belong to
        //this client.
        byte[] args =
            testProvider.getNextTest(signature);

15      //save next_test as a currentTest for
        // subsequent calls
        // currentTests.put(signature,
        if (args == null){
            httpServer.removeCurrentTest(signature);
20        }
        else{
            httpServer.setCurrentTest(signature,
                args);
        }

25      setResponseProperty("Content-Length",
        "" + (args == null ? 0 : args.length));
        sendDiagnostics(HTTP_OK, out);
        if (args != null) {
30          raw.write(args);
          raw.flush();
        }

        // cmd "get current test"
35      } else if (path.startsWith(testRoot +
        CURRENT_TEST)) {
        verboseIn(CURRENT_TEST);
        //get the current test from the bundle that
        //belongs to this client.
40      byte[] currentTest =
            httpServer.getCurrentTest(signature);

        if (currentTest != null) {
            setResponseProperty("Content-Length",
45          "" + (currentTest == null ? 0 :
            currentTest.length));
            sendDiagnostics(HTTP_OK, out);

            if (currentTest != null) {

```

39

```

        raw.write(currentTest);
        raw.flush();
    }
    }
5      else {
        throw new
          IllegalStateException
10      ("getCurrentTest called prior to the getNextTest");
    }

    // assume cmd is a request for download of
    //the JAR file to the client.
    else {

15        File file = new File(jarSourceDir, path);
        verboseIn("file download: " + file);

        if (file.isFile() && file.canRead()) {
            try {
20                FileInputStream fis = new
                    FileInputStream(file);
                DataInputStream fin = new
                    DataInputStream(fis);
                byte[] buffer = new
25                byte[(int)file.length()];
                fin.readFully(buffer);
                fin.close();
                setResponseProperty
                    ("Content-Type",
30                "application/java-archive");
                setResponseProperty("Content-Length",
                    "" + file.length());
                sendDiagnostics(HTTP_OK, out);
                raw.write(buffer);
35                raw.flush();
                return;
            } catch (IOException ioe) {
                System.out.println("error: " +
                    ioe.getMessage());
40                sendDiagnostics(HTTP_SERVER_ERROR,
                    out);
            }
        }
        sendDiagnostics(HTTP_NOT_FOUND, out);
45    }
}

/**

```



40

```

* This function processes HTTP POST requests.
* The request is retrieved from the HTTP request line.
* Then, according to the type of command
* the corresponding functionality is used.
5  */
protected void handlePost(URL request, DataInputStream in,
PrintWriter out, OutputStream raw) throws IOException {

    String ref = (request.getRef() == null) ? "" : "#"
10    + request.getRef();
    String path = request.getFile() + ref;

    if (processCommand(request, postData, out, raw, path))
    {
15        verboseIn(path + " processed");
    }
    else
        sendDiagnostics(HTTP_NOT_FOUND, out);
    }

20

/**
 * These are common actions for POST http methods
 **/
25 private boolean processCommand(URL request, byte[]
buf, PrintWriter out, OutputStream raw, String path)
throws IOException {

    // cmd "send test results"
30    if (request.getFile().startsWith(testRoot +
"sendTestResult")) {
        verboseIn("sendTestResult");

        //String signature = getSignature (path);
35        testProvider.sendTestResult(buf, signature);
        sendDiagnostics(HTTP_OK, out);
        return true;
    }
    //handle storeEventSequence cmd.
40    else if (getUIService("storeEventSequence",
path, testRoot, out, raw, buf)){
        verboseIn(path + " processed");
        return true;
    }

45    //handle showTestDescriptionString cmd.
    //For TCK tests.
    else if (getUIService
("showTestDescriptionString", path, testRoot,

```

41

```

        out, raw, buf)){
            verboseIn(path + " processed");
            return true;
        }
5
        //handle ShowCombineTestDescription cmd.
        // For J2SE tests.
        else if (getUIService("ShowCombineTestDescription",
10            path, testRoot, out, raw, buf)){
            verboseIn(path + " processed");
            return true;
        }

        //handle compareJ2SEOutputString cmd.
15        //For J2SE tests.
        else if (getUIService("compareJ2SEOutputString",
            path, testRoot, out, raw, buf)){
            verboseIn(path + " processed");
            return true;
20        }
        else
            return false;
    }

25
    /*
    * this method is called when the former task wasn't
    * ended normally, like VM_EXIT. When the next
    * "getNextApp.jad" request
30    * arrives, JDTS checks that the last task had
    * finished normally, and if doesn't then it calls
    * stopTask(), to clear the viewer in case of an UI test,
    * and calls clean up to close all the streams.
    */
35    public void stopTask(){
        try{
            if (automationManager.viewer != null) {
                automationManager.viewer.stop();
                automationManager.viewer = null;
40            }
            cleanUp();
        }catch (Exception e){
            e.printStackTrace();
45        }
    }

    /**
    * This function set to the client its unique ID. In case
    * that the IP is used is a local address (127.0.0.1),

```

```

* JDTS assigns an ID to the client, Otherwise the IP
* address IS be used as an ID, to which it is
* concatenated with another string, which may be an ID
* attached to the first client request by some outside
5 * party.
* /
    public String setSignature(String ipAddress,
        String path){
//happens only on the first time that a MIDP client
10 //connects to JDTS using the host as a "localhost"
//string.
        if (ipAddress.equals("127.0.0.1") &&
            getNextAppReq == true){
            httpServer.setLocalhostFlag();
15            String ID =
                String.valueOf(httpServer.getNextID());
                verboseLn("ID = " + ID);
                return ID;
        }
20 //last request was made using the localhost address and
//the localhost flag was set,
//That means that the requests from the client
//use a signature that was given to the JDTS server
//and there is no significance to the IP address.
25        if (httpServer.getLocalhostFlag()){
            return getAgentMonitorID(path);
        }

        //common case.
30        String agentMonitorID = getAgentMonitorID(path);
//get an ID that was made by some outside party.
        return ipAddress + "." + agentMonitorID;
    }

35    private String getAgentMonitorID(String path){
        String lastToken = null;
        StringTokenizer st =
            new StringTokenizer(path, "/");
        int tokensNum = st.countTokens();
40        while (tokensNum > 0){
            lastToken = st.nextToken();
            tokensNum--;
        }
        try{
45            //if next token is a number than the last token
            //is an ID, and an exception is not generated.
            Integer.parseInt(lastToken);
            return lastToken;
        }catch (NumberFormatException nfe){

```

43

```
        //default ID
        return "1";
    }
}

5
/**
 * A service for UI test
 */
10 private boolean getUIService(String task, String path,
    String testRoot, PrintWriter out, OutputStream raw,
    byte[] buf) throws IOException {
    try{
        synchronized(automationManager){
            verboseIn("signature = " + signature);
            verboseIn("ipAddress = " + ipAddress);
15 //set the client ID before address the UI request.
            automationManager.setSignature(signature);
            automationManager.setIPAddress(ipAddress);

20 if (task.equals("showTestDescription")){
            return automationManager.
                showTestDescription
                    (path, testRoot, out, raw);
        }
25 else if (task.equals
        ("getRecordedEventSequence")){
            return automationManager.
                getRecordedEventSequence
                    (path, testRoot, out, raw);
30 }
        else if (task.equals
            ("storeEventSequence")){
            return
                automationManager.storeEventSequence
35 (path, testRoot, out, raw, buf);
        }
        else if (task.equals
            ("showTestDescriptionString")){
            return automationManager.
                showTestDescriptionString
40 (path, testRoot, out, raw, buf);
        }
        else if (task.equals
            ("ShowCombineTestDescription")){
45 return automationManager.
                ShowCombineTestDescription
                    (path, testRoot, out, raw, buf);
        }
        else if (task.equals
```

44

```

        ("compareJ2SEOutputString")){
            return automationManager.
                compareJ2SEOutputString(path,
                    testRoot, out, raw, buf);
5          }
          else
              return false;
        }
    }catch (Exception e){
10      e.printStackTrace();
        return false;
    }
}

15

private void cleanUp(){
    try{
        /*
20      *In case that the cleanUp call made from a
        *situation of TIMEOUT, meaning that the client
        *has died, there is no need
        *for the following code
        */
25      if (status == 2){ //Client is ALIVE
        /*
        * Read and block until the end of the input
        * stream, so we know that
        * the client application got the data, not
30      * just the low level TCP.
        */
        try {
            for (; ; ) {
                if (in.read() == -1) {
35                  break;
                }
            }
        } catch (IOException se) {
            // do nothing
40        }
        in.close();
        out.close();
        raw.close();
45      conn.close();
    }catch(IOException ioe){
        ioe.printStackTrace();
    }
}

//Remove the thread 44 from the active threads list.

```

45

```
        httpServer.cleanup(signature);  
    }  
}
```

5

## Listing 6

```
/**  
 * Extender is a class that enables an agent to be linked  
10  * with platforms that require their main application  
 * class to extend their system class,  
 * such as Applet or MIDlet.  
 * Extender enables agent to delegate platform specific  
 * commands to it.  
15  *  
 * @author Victor Rosenman  
 * (SUN ISRAEL DEVELOPMENT CENTER)  
 *  
 */  
20  public interface Extender {  
    /** Enables extender to use platform specific way  
     * of application termination  
     */  
25    public void terminateAgent();  
  
    /** Retrieves a reference to platform class, which  
     * the main application class extends.  
     * The agent can provide access to the test program to  
30    * the main application class in the context  
     * in which it is running.  
     * @return an object, which can be casted to the system  
     * class, which extender class extends  
     */  
35    public Object getRunnerExtender();  
    }  
40
```